Lecture 21: Gaussian Processes and Bayesian Optimization

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

We want to optimize a function $f : \mathcal{X} \to \mathbb{R}$ over some set X (here the set $\mathcal{X}$ is the set of hyperparameters we want to search over, not the set of examples). But f is expensive to compute, making optimization difficult. Main idea of Bayesian optimization:

- Model f as a probability distribution.
- If we've computed f at parameter values $x_1, x_2, \ldots, x_D$, then we consider $f(x_1), f(x_2), \ldots, f(x_D)$ to be *observed variables* in the model.
- Any x that we haven't computed $f(x)$ for corresponds to a *hidden variable* in the model.
- Key insight: even though we haven't computed $f(x)$, the probabilistic model that we defined lets us compute the conditional distribution

  $$\mathbf{P}\left(f(x)|f(x_1), f(x_2), \ldots, f(x_D)\right).$$

  We want to choose the probabilistic model such that this is much cheaper to compute than $f(x)$ itself.
- We can use this conditional distribution to estimate $f(x)$ for values of x we haven't observed yet.
- We can also use this conditional distribution to *choose the next value of* x we are going to compute $f(x)$ at as we continue optimizing.
- Key benefit of Bayesan optimization: uses *all* the information from previous computations of $f(x)$ to choose the next point to evaluate, rather than just using information from the last or last few computations, as is done with methods like GD and Momentum.

Two major design decisions for Bayesian optimization:

- The **prior**: the probability distribution over functions that we use. This encodes our assumptions about the function f.
  - The standard way to do this is with a *Gaussian process* prior.
- The **acquision function**: how we select the next point to sample, given a conditional distribution over the values of $f(x)$.
  - Many ways to do this, as we'll see.

Review: Gaussian processes. Recall: the multivariate Gaussian distribution in d dimensions with mean $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$ has probability density function

$$\mathbf{P}\left(X\right) = \frac{1}{\sqrt{(2\pi)^k \left|\Sigma\right|}} \cdot \exp\left(-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)\right);$$

from here it is easy to check that $\mathbf{E}\left[X\right] = \mu$ and $\mathbf{E}\left[XX^T\right] = \Sigma$. This is the natural multidimensional analog of the one-dimensional Gaussian distribution you are familiar with. In order for this definition to work, we need the covariance matrix Σ to be positive semidefinite (it is, after all a covariance matrix, and

all covariance matrices are positive semidefinite). Just to see why: for any fixed vector $u \in \mathbb{R}^d$

$$u^T \Sigma u = u^T \mathbf{E}\left[XX^T\right] u = \mathbf{E}\left[u^T X X^T u\right] = \mathbf{E}\left[(X^T u)^2\right] \geq 0.$$

The average of independent vector-valued random variables tends to converge towards a multivariate Gaussian distribution in the same way that the average of independent scalar-valued random variables tends to converge towards a Gaussian distribution in one dimension. This is called the *Multidimensional Central Limit Theorem* and it's a natural extension of the 1-D CLT you all know and love. When we are doing hyperparameter optimization to minimize some empirical risk, that empirical risk evaluated at many different points is the average of a bunch of component loss functions. That is,

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_D) \end{bmatrix} = \frac{1}{n} \sum_{i=1}^{n} \begin{bmatrix} f_i(x_1) \\ \vdots \\ f_i(x_D) \end{bmatrix}$$

so it's natural to suspect that this vector of empirical risk values would be approximately Gaussian distributed. So, we'd like to use a multidimensional Gaussian distribution as the prior for Bayesian optimization. The problem: there are usually infinitely many possible settings for the hyperparameters (i.e. $\mathcal{X}$ is an infinite set in most cases). And multivariate Gaussian distributions assume a *finite* number of dimensions. The solution to this is to use what's called a **Gaussian process**: this is the natural infinite-dimensional analog of the multidimensional Gaussian. Typically, we use the all-zeros vector for the mean μ, and replace the covariance matrix Σ with a Kernel function K.[1] If f is the function we're modeling, then this is equivalent to assuming that for any x_i and $x_j \in \mathcal{X}$,

$$\mathbf{E}\left[f(x_i)\right] = 0 \qquad \text{and} \qquad \mathbf{E}\left[f(x_i) \cdot f(x_j)\right] = K(x_i, x_j).$$

Importantly, here this expected value represents an expectation over our *belief* about what the value of the function f is, not an expectation over the random sampling of the dataset or any randomness that may exist in our training algorithm.

Why does this function have to be a kernel? Recall that kernels had two properties: symmetry and positive semidefiniteness. The symmetry here is needed because the expression $f(x_i) \cdot f(x_j)$ is symmetric. And the positive semidefiniteness is needed because any covariance matrix must be positive semidefinite (even, it turns out, if it is infinite-dimensional).

Important property of Gaussian processes. The marginal distribution of a finite number of variables of a Gaussian process is a multivariate Gaussian distribution. That is, if f is a Gaussian process, then for any $x_1, x_2, \ldots, x_D \in \mathcal{X}$

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_D) \end{bmatrix} \quad \text{is multivariate-Gaussian-distributed with mean } \mu = 0$$

$$\text{and covariance } \Sigma = \begin{bmatrix} K(x_1, x_1) & K(x_1, x_2) & \cdots & K(x_1, x_D) \\ \vdots & \vdots & \ddots & \vdots \\ K(x_D, x_1) & K(x_D, x_2) & \cdots & K(x_D, x_D) \end{bmatrix}. \tag{1}$$

In particular, this lets us find the marginal distribution of $f(x_*)$ at a new test value x_* conditioned on the values of the function f we've already observed. Explicitly, if we define Σ as above in (1), define the vector $\mathbf{k}_*$ as

$$\mathbf{k}_* = \begin{bmatrix} K(x_1, x_*) & K(x_2, x_*) & \cdots & K(x_D, x_*) \end{bmatrix}^T,$$

[1] Now you see why I brought kernels back up in the last lecture.

and if $y = \begin{bmatrix} y_1 & y_2 & \cdots & y_D \end{bmatrix}^T$ is the vector of observations we've made so far then we can write the marginal distribution of $f(x_*)$ as

$$f(x_*)|(f(x_1) = y_1, f(x_2) = y_2, \cdots, f(x_D) = y_D) \sim \mathcal{N}(\mathbf{k}_*^T \Sigma^{-1} y, K(x_*, x_*) - \mathbf{k}_*^T \Sigma^{-1} \mathbf{k}_*)$$

where $\mathcal{N}(\mu, \sigma^2)$ denotes the normal distribution with mean μ and variance σ^2.

What is the computational cost of computing the mean and variance of these marginal distributions for N different test points x_*? How much memory is needed? Suppose that the kernel function K takes $\Theta(d)$ time to compute (for some constant d).

Gaussian processes as a prior for Bayesian optimization. To use a Gaussian process for Bayesian optimization, just let the domain of the Gaussian process $\mathcal{X}$ be the space of hyperparameters, and define some kernel that you believe matches the similarity of two hyperparameter assignments. Typically, you want to set the kernel based on your intuition about the problem. You can also set it based on prior work in the literature...as usual the kernel can become another hyperparameter (or hyper-hyperparameter) you need to set. Then the Gaussian process can be used as a prior for the observed and unknown values of the loss function f (as a function of the hyperparameters).

Bayesian optimization.

Algorithm 1 Bayesian optimization with Gaussian process prior

input: loss function f, kernel K, acquisition function a, loop counts N_{warmup} and N
▷ warmup phase
$y_{\text{best}} \leftarrow \infty$
for $i = 1$ **to** N_{warmup} **do**
  select x_i via some method (usually random sampling)
  compute exact loss function $y_i \leftarrow f(x_i)$
  if $y_i \leq y_{\text{best}}$ **then**
    $x_{\text{best}} \leftarrow x_i$
    $y_{\text{best}} \leftarrow y_i$
  end if
end for
for $i = N_{\text{warmup}} + 1$ **to** N **do**
  update kernel matrix $\Sigma \in \mathbb{R}^{i \times i}$ according to (1)
  let $\mu(x_*)$ and $\sigma(x_*)$ denote the expected value and standard deviation, respectively, of $f(x_*)$ under the Gaussian process model, conditioned on all the previous observations of $f(x_i) = y_i$
  $x_i \leftarrow \arg\min_{x_*} a(\mu(x_*), \sigma(x_*), y_{\text{best}})$
  compute exact loss function $y_i \leftarrow f(x_i)$
  if $y_i \leq y_{\text{best}}$ **then**
    $x_{\text{best}} \leftarrow x_i$
    $y_{\text{best}} \leftarrow y_i$
  end if
end for
return x_{best}

The acquisition function. Determines how we search for new points. There are a few common acquisision functions that are used for machine learning. For a Gaussian process prior, they are generally a function of three things: the mean of the hidden variable $f(x_*)$, the standard deviation of $f(x_*)$, and the best value seen so far during optimization, y_{best}.

Probability of improvement. The probability of improvement (PI) acquisition function asks us to maximize the probability that we will observe an improvement from the next point searched. That is, it tries to maximize the probability that the x_i that we test will be our new "best" x_i. This probability is

$$\mathbf{P}\left(f(x_*) < y_{\text{best}}\right) = \mathbf{P}\left(\frac{f(x_*) - \mu(x_*)}{\sigma(x_*)} < \frac{y_{\text{best}} - \mu(x_*)}{\sigma(x_*)}\right) = \Phi\left(\frac{y_{\text{best}} - \mu(x_*)}{\sigma(x_*)}\right),$$

where Φ is the cumulative distribution function of the standard Gaussian distribution. Since we're trying to maximize this, we can use the negation of this in the Bayesian optimization algorithm as an activation function (alternatively, we could not negate and just maximize instead of minimize in the Bayesian optimization algorithm description—it's equivalent).

$$a_{\text{PI}}(y_{\text{best}}, \mu, \sigma) = -\Phi\left(\frac{y_{\text{best}} - \mu}{\sigma}\right)$$

Expected improvement. The expected improvement (EI) acquisition function asks us to minimize the expected improvement in the value of the new y_{best} after the next point is searched. This expected value is

$$\mathbf{E}\left[\min(f(x_*) - y_{\text{best}}, 0)\right] = \mathbf{E}\left[\min\left(\frac{f(x_*) - \mu(x_*)}{\sigma(x_*)} - \frac{y_{\text{best}} - \mu(x_*)}{\sigma(x_*)}, 0\right)\right] \cdot \sigma(x_*).$$

To simplify this further, we need an expression for $\mathbf{E}_u\left[\min(u - c, 0)\right]$ when u is standard-Gaussian-distributed i.e. $u \sim \mathcal{N}(0, 1)$. We can get this by solving the integral, where ϕ is the probability distribution function of the Gaussian distribution.

$$\mathbf{E}_u\left[\min(u - c, 0)\right] = \int_{-\infty}^{c} (u - c) \cdot \phi(u)\, du = \left[-\phi(u) - c \cdot \Phi(u)\right]_{-\infty}^{c} = -\phi(c) - c \cdot \Phi(c).$$

It follows that we can let our acquisition function be

$$a_{\text{EI}}(y_{\text{best}}, \mu, \sigma) = -\left(\phi\left(\frac{y_{\text{best}} - \mu}{\sigma}\right) + \frac{y_{\text{best}} - \mu}{\sigma} \cdot \Phi\left(\frac{y_{\text{best}} - \mu}{\sigma}\right)\right) \cdot \sigma.$$

Once again this is an exact expression we can compute easily from μ, σ, and y_{best}.

Lower confidence bound. Another common type of activation function is the lower confidence bound activation function, which is designed to minimize *regret* over the course of optimization. For some parameter κ (not the condition number!) this activation function is defined as

$$a_{\text{LCB}}(y_{\text{best}}, \mu, \sigma) = \mu - \kappa \cdot \sigma.$$

Here, the parameter κ trades off between exploration and exploitation. A small κ leads to more exploitation, whereas a large κ explores more high-variance points at which less is known about the value of the function.

To think about as we leave off: what are the main computational costs of hyperparameter optimization? Which aspects of the system still need to be determined before we can actually run this algorithm in practice?

Despite its nice properties, we don't always use Bayesian optimization all the time. **Why?**

- Sensitivity to choice of kernel—need to choose good hyperparameters for kernel
- Need to solve the inner optimization problem efficiently somehow—not clear how to do this
- Difficulty with scaling—scary cubic terms in the computational complexity
- Default setup doesn't handle varying cost to compute objective—but we can see substantial variation here in practice, especially when we are exploring systems parameters

Kernel selection. Choosing a good prior is very important for the performance of Bayesian optimization. We want to choose a kernel that is effective for machine learning. We could just use the RBF kernel,

$$K_{\text{RBF}}(x, y) = \exp\left(\gamma \cdot \|x - y\|^2\right).$$

But imagine we were trying to optimize, say, the momentum and the number of hidden units in a layer. **What problem might we have?**

One way to address hyperparameter scaling is to use the *automatic relevance determination (ARD) squared-exponential* kernel. This is defined, for $x, y \in \mathbb{R}^d$, as

$$K_{\text{ARD}}(x, y) = \theta_0 \cdot \exp\left(-\frac{1}{2} r^2(x - y)\right) \qquad \text{where} \qquad r^2(x - y) = \sum_{i=1}^{d} \frac{(x_i - y_i)^2}{\theta_i^2}.$$

Here, $\theta_0, \theta_1, \ldots, \theta_d$ are hyperparameters of the kernel (we can think of these as hyper-hyperparameters). **How does this help with hyperparameter scaling?**

For many tasks, the ARD kernel makes predictions that are too smooth to accurately match the true loss function f. To address this, it's common to use the **Matèrn kernel**, which allows for less smooth predictions. It's usually written as

$$K_{M(\nu)}(x, y) = \sigma^2 \cdot \frac{2^{1-\nu}}{\Gamma(\nu)} \cdot \left(\frac{2\nu r^2(x - y)}{\rho^2}\right)^{\frac{\nu}{2}} K_\nu \left(\sqrt{\frac{2\nu r^2(x - y)}{\rho^2}}\right),$$

where K_ν denotes the modified Bessel function of the second kind, and the parameters for this kernel are ν, σ^2, ρ, and the θ_i from the definition of r above. But this is super messy, so we normally fix a value of ν (this determines how rough our predictions will be) which allows us to simplify the definition substantially. It's common to choose $\nu = 5/2$, in which case the **ARD Matèrn 5/2 kernel** is given by

$$K_{M(5/2)}(x, y) = \theta_0 \cdot \left(1 + \sqrt{5r^2(x - y)} + \frac{5}{3} r^2(x - y)\right) \cdot \exp\left(-\sqrt{5r^2(x - y)}\right).$$

Here, the hyperparameters are $\theta_0, \theta_1, \ldots, \theta_d$ as before for the ARD kernel. Using this lets us both solve the hyperparameter scaling problem and avoid making predictions that are too smooth.

Choosing the hyper-hyperparameters. One problem with what we just did: now we have a good kernel that we can use for general problems, but we just added in some hyper-hyperparameters that we need to set (the θ_i). Worse, there's *more* hyper-hyperparameters than we had hyperparameters, so the problem seems to have gotten harder! There are a couple of standard ways to try to automatically set these hyper-hyperparameters.

One way to do it is to use **maximum likelihood estimation**. We collect a sample of points from f (usually this is our random sampling we used to warm up the Gaussian process) and find the hyper-hyperparameters θ

that maximize the probability of observing those points in the Gaussian process model. That is, if we observe f at D different points $x_1, \ldots, x_D$ and observe it having values $y_1, \ldots, y_D$, then we select the parameters as

$$\theta = \arg\max_{\theta} \mathbf{P}\left(f(x_1) = y_1, f(x_2) = y_2, \ldots, f(x_D) = y_D\right)$$

where this probability is taken within the Gaussian process model with parameters θ.

Solving the inner optimization problem. To run Bayesian optimization, recall that we needed to solve the optimization problem

$$\begin{aligned} \text{minimize: } & a(\mu(x_*), \sigma(x_*), y_{\text{best}}) \\ \text{over: } & x_* \in \mathcal{X}. \end{aligned}$$

How do we solve this? Some ways to do it:

- We usually can differentiate a, so gradient descent is an option. But a is usually non-convex so this can be tricky.
- Common approach: choose a random starting point and run gradient descent until it converges. Then choose another random starting point and repeat many times.

Modeling costs. Hypothetical scenario: suppose that the amount of time it takes to compute f varies by an order of magnitude for different hyperparameters x. We want to minimize $f(x)$ while not spending too much time on computing f.

A new acquisition function for this setting: **expected improvement per-second**. Idea: model not only the value of f but also the time it will take to compute f as a Gaussian process (i.e. we have *two* Gaussian processes here). If $c(x_*)$ denotes the predicted compute cost for computing $f(x_*)$ from our Gaussian process, then the expected improvement per second is

$$\mathbf{E}\left[\frac{\min(f(x_*) - y_{\text{best}}, 0)}{c(x_*)}\right]$$

which leads naturally to an acquisition function like

$$a_{\text{EI}/s}(y_{\text{best}}, \mu, \sigma, x_*) = -\left(\phi\left(\frac{y_{\text{best}} - \mu}{\sigma}\right) + \frac{y_{\text{best}} - \mu}{\sigma} \cdot \Phi\left(\frac{y_{\text{best}} - \mu}{\sigma}\right)\right) \cdot \frac{\sigma}{\mathbf{E}\left[c(x_*)\right]}$$

What else can we do to incorporate cost into our hyperparameter optimization?